Making the Point-free Calculus Less Pointless

Alcino Cunha Jorge Sousa Pinto

Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal
{alcino,jsp}@di.uminho.pt

Functional programming is particularly well suited for equational reasoning – referential transparency ensures that expressions in functional programs behave as ordinary expressions in mathematics. However, unstructured programming can still difficult formal treatment. As such, when John Backus proposed a new functional style of programming in his 1977 ACM Turing Award lecture, the main features were the absence of variables and the use of *functional forms* or *combinators* to combine existing functions into new functions [1]. The choice of the combinators was based not only on their programming power, but also on the power of the associated algebraic laws. Quoting Backus: *"Associated with the functional style of programming is an algebra of programs [...] This algebra can be used to transform programs and to solve equations whose "unknowns" are programs in much the same way one transforms equations in high-school algebra"*.

This style of programming is usually called *point-free*, as opposed to the *point-wise* style, where the arguments are explicitly stated. The basic set of combinators used in this paper as been already extensively presented in many publications, such as [6], and includes the typical products, with split $(\cdot \vartriangle \cdot)$ and projections fst and snd, sums, with either $(\cdot \triangledown \cdot)$ and injections inl and inr, and exponentials, with curry $\overline{\cdot}$ and application ap.

Although the point-free style has a rich calculus for reasoning about programs, there are still many authors that resort to the point-wise style both for programming and for calculation. They claim that the point-free style is not very natural since the intuitive meaning of programs can easily be lost, and jokingly call it the *pointless* style. In fact, we agree that some point-free derivations are very long and tedious, namely when dealing with higher-order functions. However, we don't think this is an intrinsic disadvantage of point-free, but merely lack of adequate combinators and proof methodology.

As such, the objective of this work is to improve the machinery that is used to perform point-free calculations, namely in a higher-order setting. As Jeremy Gibbons puts it [3], *"We are interested in extending what can be calculated precisely because we are not interested in the calculations themselves [...]"*, or, in other words, we aim at extending the calculus with new useful operators that help reducing the burden of proofs just to the creative parts.

Point-free programming is usually complemented with extensive use of *recursion patterns* – higher-order operators that encapsulate typical patterns of recursion, such as the well-known *fold* or *catamorphism*. They prevent the use of arbitrary recursive definitions, and also have a nice set of equational laws. Although initially they were only defined for lists, it became clear that they could be generalized for any recursive data type viewed as fixed point of a functor [5]. In this paper we will only use the catamorphism, that given a function of type $g : F\ A \to A$ is denoted by $(\!|g|\!)_F : \mu F \to A$, the function that builds its result by replacing the constructors of the input by g. One of the most important laws about this recursion operator is fusion – given a strict f, if $f \circ g = h \circ Ff$ then $f \circ (\!|g|\!)_F = (\!|h|\!)_F$.

Suppose that we want to obtain a efficient version of the reverse function using the *accumulation strategy* introduced by Richard Bird [2]. This technique uses fusion to derive it from an inefficient version using the concatenation operator $\mathsf{cat} : \mathsf{List}\ A \times \mathsf{List}\ A \to \mathsf{List}\ A$.

$$\begin{array}{lcl} \mathsf{reverse} & : & \mathsf{List}\ A \to (\mathsf{List}\ A \to \mathsf{List}\ A) \\ \mathsf{reverse} & = & \overline{\mathsf{cat}} \circ (\!|\underline{\mathsf{nil}} \triangledown (\mathsf{cat} \circ \mathsf{swap} \circ (\mathsf{wrap} \times \mathsf{id}))|\!) \end{array}$$

In order to perform this derivation we must use the associativity property of cat, that is usually expressed in point-free as $\mathsf{cat} \circ (\mathsf{id} \times \mathsf{cat}) \circ \mathsf{assocr} = \mathsf{cat} \circ (\mathsf{cat} \times \mathsf{id})$. Unfortunately, this formulation is not very practical because we need refer to the operator in curried form, and its use would lead to a lengthy derivation. A simpler calculation can be obtained if an uncurried version of the composition

combinator was available. This trick was already mentioned in some publications, such as [4], but it is usually defined in point-wise style which invalidates a pure point-free calculus. Instead, we define it as follows.

$$\begin{array}{lcl} \mathsf{comp} & : & (C^B \times B^A) \to C^A \\ \mathsf{comp} & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times \mathsf{ap}) \circ \mathsf{assocr}} \end{array}$$

Using comp we can express associativity as $\overline{\oplus} \circ \oplus = \mathsf{comp} \circ (\overline{\oplus} \times \overline{\oplus})$. Now we can easily derive by fusion the implementation of reverse using an accumulation parameter, obtaining $(\!|\underline{\mathsf{id}} \triangledown (\mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{cons}} \times \mathsf{id}))|\!)$.

Consider now the function that determines the initial sums of a list. Again, we can derive an optimized version using the accumulation technique and fusion. Taking $\oplus = \mathsf{ap} \circ ((\mathsf{List} \circ \overline{\mathsf{plus}}) \times \mathsf{id}) \circ \mathsf{swap}$ we have the following specification.

$$\begin{array}{lcl} \mathsf{isums} & : & \mathsf{List\ Nat} \to (\mathsf{Nat} \to \mathsf{List\ Nat}) \\ \mathsf{isums} & = & \overline{\oplus} \circ (\!|\underline{\mathsf{nil}} \triangledown (\oplus \circ \mathsf{swap} \circ (\mathsf{id} \times (\mathsf{cons} \circ (\underline{\mathsf{zero}} \vartriangle \mathsf{id}))))|\!) \end{array}$$

Similarly, we can simplify the derivation if we use an uncurried version of the split combinator.

$$\begin{array}{lcl} \mathsf{split} & : & (B^A \times C^A) \to (B \times C)^A \\ \mathsf{split} & = & \overline{(\mathsf{ap} \times \mathsf{ap}) \circ ((\mathsf{fst} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))} \end{array}$$

Equipped with split we can redefine $\oplus \circ (\mathsf{cons} \times \mathsf{id}) = \mathsf{cons} \circ (\mathsf{plus} \times \oplus) \circ ((\underline{\mathsf{fst}} \times \mathsf{id}) \vartriangle (\mathsf{snd} \times \mathsf{id}))$, a property of $\oplus$ need for the calculation, simply as $\overline{\oplus} \circ \mathsf{cons} = \mathsf{cons}^\bullet \circ \mathsf{split} \circ (\overline{\mathsf{plus}} \times \overline{\oplus})$. The resulting accumulation is $(\!|\underline{\underline{\mathsf{nil}}} \triangledown (\mathsf{comp} \circ \mathsf{swap} \circ (\overline{\mathsf{plus}} \times (\mathsf{cons}^\bullet \circ \mathsf{split} \circ (\underline{\mathsf{id}} \vartriangle \mathsf{id}))))|\!)$.

Further complications arise if we want to apply the accumulation technique to derive functions with two accumulating parameters. For example, a tail recursive function to compute the height of a leaf tree, whose data type is defined as $\mathsf{LTree}\ A = \mu(\underline{A} + \mathsf{Id} \times \mathsf{Id})$, can be derived from the following specification.

$$\begin{array}{lcl} \mathsf{height} & : & \mathsf{LTree}\ A \to (\mathsf{Nat} \to (\mathsf{Nat} \to \mathsf{Nat})) \\ \mathsf{height} & = & \overline{\mathsf{max}}^\bullet \circ \overline{\mathsf{plus}} \circ (\!|\underline{\mathsf{zero}} \triangledown (\mathsf{succ} \circ \mathsf{max})|\!) \end{array}$$

This calculation can be simplified by introducing the following left-exponentiation operator, a kind of dual to the normal exponentiation, that enjoys nice equational laws like $f^\bullet \circ {}^\bullet g = {}^\bullet g \circ f^\bullet$.

$$\begin{array}{lcl} {}^\bullet f & : & A^C \to A^B \\ {}^\bullet f & = & \overline{\mathsf{ap} \circ (\mathsf{id} \times f)} \end{array}$$

The resulting tail-recursive height is $(\!|\overline{\underline{\mathsf{max}}} \triangledown {}^\bullet\mathsf{succ} \circ \mathsf{comp}^\bullet \circ \mathsf{split}|\!)$. We agree that the resulting catamorphisms are cumbersome, but converting them to point-wise is straightforward. We suggest the reader to perform that task, in order to check that the optimized functions behave as expected.

We have already developed a library that enables programming in Haskell in a true point-free style (see `http://wiki.di.uminho.pt/twiki/bin/view/Alcino/PointlessHaskell`). Using this library, we intend to develop an automatic transformation system for point-free programs using term rewriting, hoping to prove also in practice the advantages of the point-free calculus over the point-wise one.

References

[1] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

[2] Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

[3] Jeremy Gibbons. An introduction to the bird-meertens formalism. In *New Zealand Formal Program Development Colloquium Seminar*, Hamilton, November 1994.

[4] Jeremy Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[5] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, October 1990.

[6] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.